A Fast Mixed-Precision Strategy for Iterative GPU-Based Solution of the Laplace Equation

Glimberg, Stefan Lemvig

Publication date:
2011

Document Version
Publisher's PDF, also known as Version of record

A Fast Mixed-Precision Strategy for Iterative GPU-Based Solution of the Laplace Equation

Stefan L. Glimberg

Section of Scientific Computing
Department of Informatics and Mathematical Modelling
Technical University of Denmark

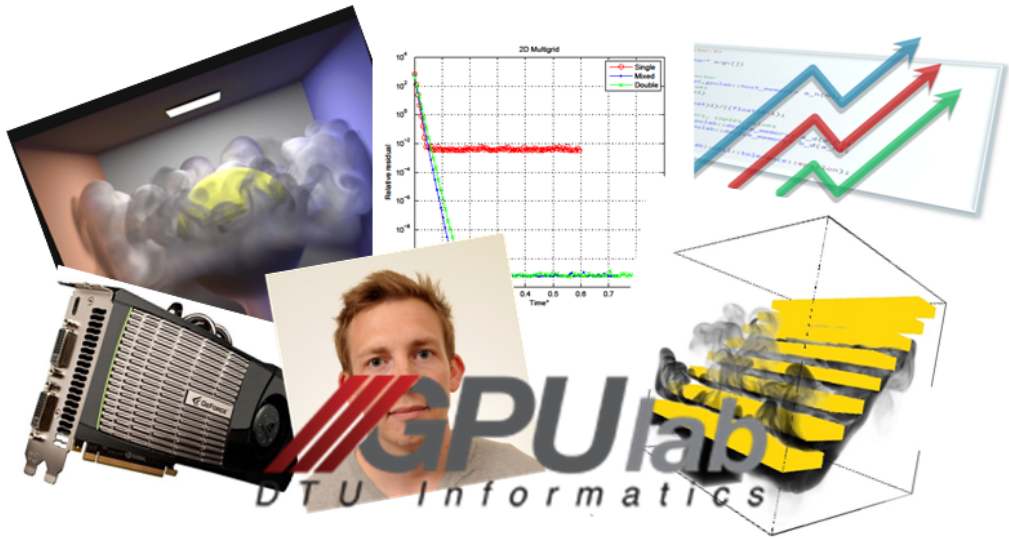
Argonne National Laboratory
November 29^{th}, 2011

GPUlab DTU Informatics

Background

Stefan L. Glimberg

- PhD student, started 2010
- Technical University of Denmark - Section of Scientific Computing
- Project: *Scientific GPU Computing for PDE Solvers*
- Visiting UIUC this semester



GPUlab DTU Informatics

http://gpulab.imm.dtu.dk/

The GPUlab is a competence center and laboratory for the use of Graphics Processing Units (GPUs) for visualization, scientific computations, and high-performance computing. The purpose is to attract focal interests in the use of GPUs by both engineering students and researchers in projects.

Projects

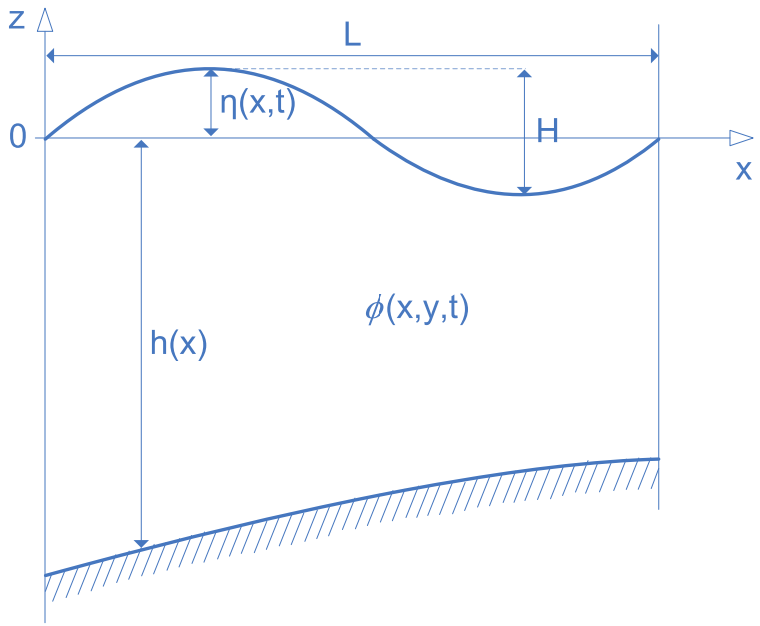
- Auto-tuning of Dense Linear Algebra on GPUs
- Accelerating Economic Model Predictive Control using GPUs
- Fast simulation of fully nonlinear water waves
- ...
- Your project?

A Fast Mixed-precision Strategy for Iterative GPU-based Solution of the Laplace Equation

Fully Nonlinear Free Surface Water Waves

The potential flow equations describe fully nonlinear water waves under the assumption of inviscid and irrotational flow.

2D Potential Flow Equations



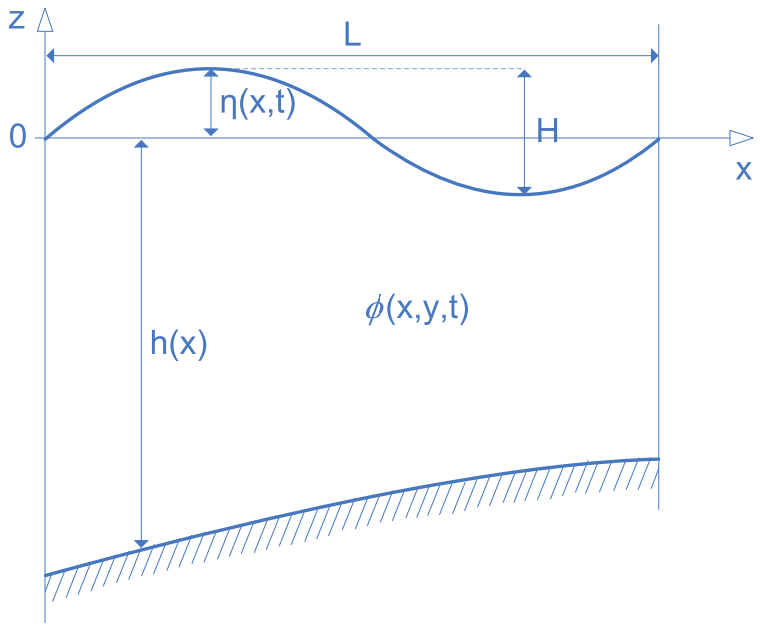
Wave parameters

- η - surface elevation
- ϕ - potential ($u = \nabla\phi$)
- h - still water depth
- $k = 2\pi/L$ - wave number
- kh - dispersion
- H/L - nonlinearity

Fully Nonlinear Free Surface Water Waves

The potential flow equations describe fully nonlinear water waves under the assumption of inviscid and irrotational flow.

2D Potential Flow Equations



$$\partial_t \eta = -\partial_x \eta \, \partial_x \tilde{\phi} + \tilde{\omega}(1 + (\partial_x \eta)^2)$$

$$\partial_t \tilde{\phi} = -g\eta - \frac{1}{2}((\partial_x \tilde{\phi})^2 - \tilde{\omega}^2(1 + (\partial_x \eta)^2))$$

$$\tilde{\omega} = \partial_z \tilde{\phi}, \quad \tilde{\phi} = \phi|_{z=\eta}$$

For $\tilde{\omega}$ to be computed, we need to know the potential in the entire domain.

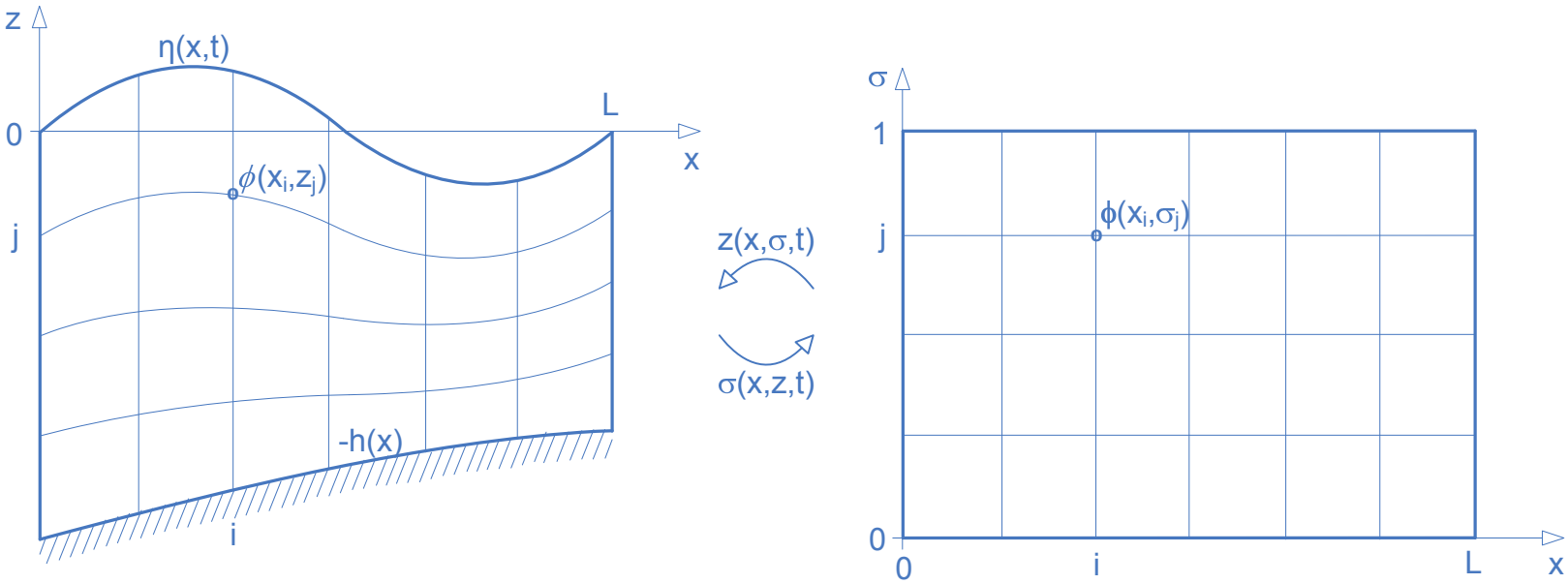
$$\phi = \tilde{\phi}, \quad z = \eta$$

$$\partial_{xx}\phi + \partial_{zz}\phi = 0, \quad -h \leq z < \eta$$

$$\partial_z \phi + \partial_x h \, \partial_x \phi = 0, \quad z = -h$$

σ-Transformed Laplace Equation

$$\sigma(x,z,t) = \frac{z + h(x)}{\eta(x,t) + h(x)}$$



$$\Phi = \tilde{\phi}, \quad \sigma = 1$$

$$\partial_{xx}\Phi + \partial_{xx}\sigma(\partial_\sigma\Phi) + 2\partial_x\sigma(\partial_{x\sigma}\Phi) + ((\partial_x\sigma)^2 + (\partial_z\sigma)^2)\partial_{\sigma\sigma}\Phi = 0, \quad 0 \leq \sigma < 1$$

$$(\partial_z\sigma + \partial_x h\partial_x\sigma)\partial_\sigma\Phi + \partial_x h\partial_x\Phi = 0, \quad \sigma = 0$$

Linear Free Surface Water Waves

If wave amplitudes are small $\eta < \epsilon$, then the total water depth is almost the same as the still water depth ($\eta + h \approx h$). If also the derivatives in η and h are assumed to be zero, the free surface equations take linear form.

Linearized Laplace Equation

$$\begin{aligned} \Phi &= \tilde{\phi}, & \sigma &= 1 \\ \partial_{xx}\Phi + (\partial_z \sigma)^2 \partial_{\sigma\sigma}\Phi &= 0, & 0 \leq \sigma &< 1 \\ \partial_z \sigma \, \partial_\sigma \Phi &= 0, & \sigma &= 0 \end{aligned}$$

These equations might serve as an approximation for the fully nonlinear equations and can thus be used for preconditioning.

A Fast Mixed-precision Strategy for Iterative GPU-based Solution of the Laplace Equation

Motivation for GPU computing

There are several good reasons to consider Graphical Processing Units for high-performance computing

- Massively parallel architecture, $\sim$ 500 cores.
- Teraflops of floating point performance
- Moderate prices $\$100 - \$2,000$. A personal super computer
- Fairly easy to get started (CUDA, OpenCL)
- Number 2 and 4 on top500 are based on GPUs

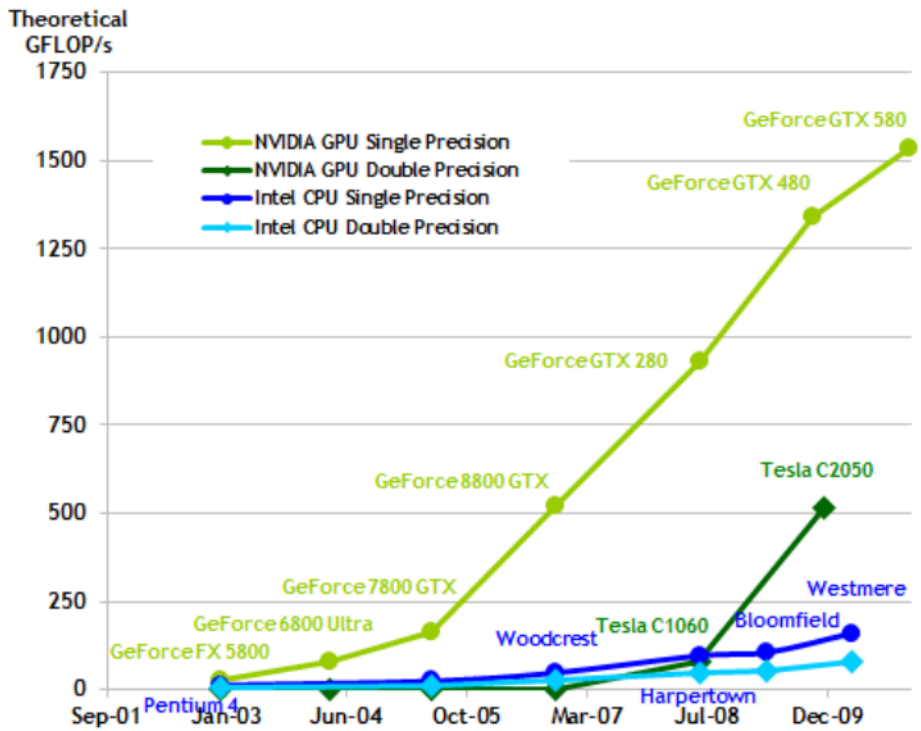


Figure: Theoretical peak performance of CPUs vs GPUs within recent years.

Motivation for GPU computing

There are several good reasons to consider Graphical Processing Units for high-performance computing

- Massively parallel architecture, $\sim$ 500 cores.
- Teraflops of floating point performance
- Moderate prices $\$100 - \$2,000$. A personal super computer
- Fairly easy to get started (CUDA, OpenCL)
- Number 2 and 4 on top500 are based on GPUs

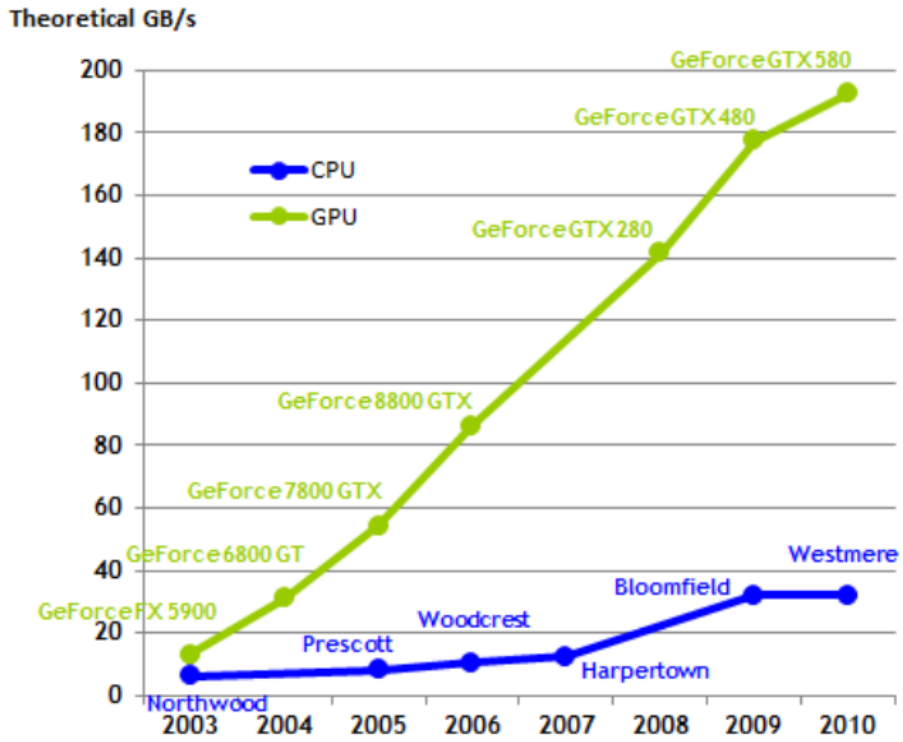


Figure: Theoretical memory throughput of CPUs vs GPUs within recent years.

Motivation for GPU computing

There are several good reasons to consider Graphical Processing Units for high-performance computing

- Massively parallel architecture, $\sim$ 500 cores.
- Teraflops of floating point performance
- Moderate prices $\$100 - \$2,000$. A personal super computer
- Fairly easy to get started (CUDA, OpenCL)
- Number 2 and 4 on top500 are based on GPUs

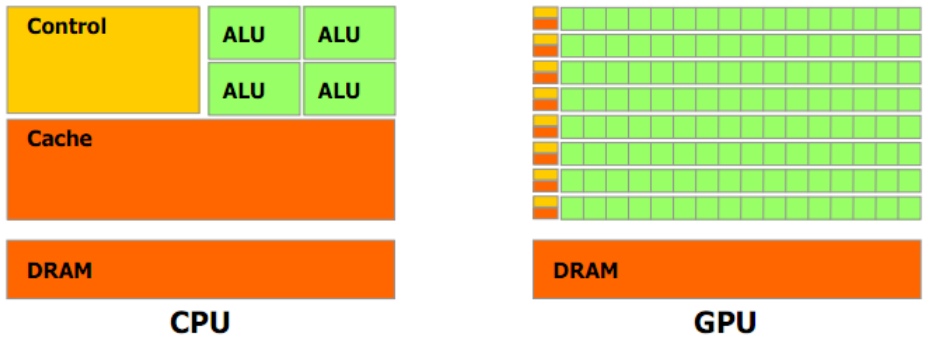


Figure: Rough sketch of the chip transistor layout for a CPU vs a GPU.

CUDA Implementation Example

Implementing a simple CUDA program is not very difficult.

1. Familiarize yourselves with CUDA syntax/keywords
2. Localize parts in the code that can be parallelized
3. Execute a lot of threads, each processing one element

CUDA Implementation Example

Implementing a simple CUDA program is not very difficult.

1. Familiarize yourselves with CUDA syntax/keywords
2. Localize parts in the code that can be parallelized
3. Execute a lot of threads, each processing one element

BLAS1 Example: $\mathbf{y} = a\mathbf{x} + \mathbf{y}$

Host (CPU):

```
1  void
2  axpy_host(float a, float* x, float
       * y, int N)
3  {
4    for(int i=0; i<N; ++i)
5    {
6      y[i] = a*x[i] + y[i];
7    }
8  }
```

CUDA Implementation Example

Implementing a simple CUDA program is not very difficult.

1. Familiarize yourselves with CUDA syntax/keywords
2. Localize parts in the code that can be parallelized
3. Execute a lot of threads, each processing one element

BLAS1 Example: $\mathbf{y} = a\mathbf{x} + \mathbf{y}$

Host (CPU):

```
1  void
2  axpy_host(float a, float* x, float
       * y, int N)
3  {
4    for(int i=0; i<N; ++i)
5    {
6      y[i] = a*x[i] + y[i];
7    }
8  }
```

Device (GPU):

```
1  __global__ void
2  axpy_device(float a, float* x,
       float* y, int N)
3  {
4    int i = blockDim.x*blockIdx.x+
         threadIdx.x;
5    y[i] = a*x[i] + y[i];
6  }
```

CUDA Implementation Example

Implementing a simple CUDA program is not very difficult.

1. Familiarize yourselves with CUDA syntax/keywords
2. Localize parts in the code that can be parallelized
3. Execute a lot of threads, each processing one element

BLAS1 Example: $\mathbf{y} = a\mathbf{x} + \mathbf{y}$

Host (CPU):

```
1  void
2  axpy_host(float a, float* x, float
       * y, int N)
3  {
4    for(int i=0; i<N; ++i)
5    {
6      y[i] = a*x[i] + y[i];
7    }
8  }
```

Better one:

```
1  template <typename T>
2  __global__ void
3  axpy_device(T a, T* x, T* y, int N
       )
4  {
5    int i = blockDim.x*blockIdx.x+
         threadIdx.x;
6    y[i] = a*x[i] + y[i];
7  }
```

CUDA Implementation Example

Implementing a simple CUDA program is not very difficult.

1. Familiarize yourselves with CUDA syntax/keywords
2. Localize parts in the code that can be parallelized
3. Execute a lot of threads, each processing one element

BLAS1 Example: $\mathbf{y} = a\mathbf{x} + \mathbf{y}$

Host (CPU):

```
1  void
2  axpy_host(float a, float* x, float
       * y, int N)
3  {
4    for(int i=0; i<N; ++i)
5    {
6      y[i] = a*x[i] + y[i];
7    }
8  }
```

Better one:

```
1  template <typename T>
2  __global__ void
3  axpy_device(T a, T* x, T* y, int N
       )
4  {
5    int i = blockDim.x*blockIdx.x+
         threadIdx.x;
6    y[i] = a*x[i] + y[i];
7  }
```

However, converting entire solvers for engineering applications is difficult, and it is even more difficult to get the best possible performance.

GPUlab DTU Informatics

A GPU-based Framework for PDE Solvers

We have build a highly generic heterogenous CPU-GPU framework for fast PDE solver prototyping (Inspired by PETSc).

Framework Objectives

- Remove all GPU-specific code for the non-expert GPU programmer
- While maintaining the possibility to customize code at kernel level

```
1  gpulab::vector<float,host_memory>    x_h(100,3.f); // Create host vector x, size 100, value 3
2  gpulab::vector<float,device_memory>  x_d(x_h);     // Create device vector x, transfer host data
3  gpulab::vector<float,device_memory>  y_d(x_d);     // Create device vector y, copy device data
4  y_d.axpy(4.f,x_d);                                 // Do y = a*x+y on the device
5  y_d.nrm2();                                        // Calculate the 2-norm on the device
```

Implementations are partly based on Thrust – a high-level interface for GPU programming.

GPUlab DTU Informatics

A Finite Difference Example

Based on Taylor series expansion we can derive a set of coefficients for calculating any derivative of u:

$$\frac{\partial^p u(x_i)}{\partial x^p} \approx \sum_{n=-\alpha}^{\beta} c_n u(x_{i+n})$$

For given p; α, β and the coefficients c_n can be determined. If $\alpha = \beta = 1$ the corresponding finite difference matrix becomes

$$\begin{bmatrix} c_{00} & c_{01} & c_{02} & 0 & 0 & 0 & 0 & 0 \\ c_{10} & c_{11} & c_{12} & 0 & 0 & 0 & 0 & 0 \\ 0 & c_{10} & c_{11} & c_{12} & 0 & 0 & 0 & 0 \\ 0 & 0 & c_{10} & c_{11} & c_{12} & 0 & 0 & 0 \\ 0 & 0 & 0 & c_{10} & c_{11} & c_{12} & 0 & 0 \\ 0 & 0 & 0 & 0 & c_{10} & c_{11} & c_{12} & 0 \\ 0 & 0 & 0 & 0 & 0 & c_{10} & c_{11} & c_{12} \\ 0 & 0 & 0 & 0 & 0 & c_{20} & c_{21} & c_{22} \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \end{bmatrix} \approx \begin{bmatrix} \partial^p u(x_0)/\partial x^p \\ \partial^p u(x_1)/\partial x^p \\ \partial^p u(x_2)/\partial x^p \\ \partial^p u(x_3)/\partial x^p \\ \partial^p u(x_4)/\partial x^p \\ \partial^p u(x_5)/\partial x^p \\ \partial^p u(x_6)/\partial x^p \\ \partial^p u(x_7)/\partial x^p \end{bmatrix}$$

There is a lot of repetitions in the matrix and it is very sparse.

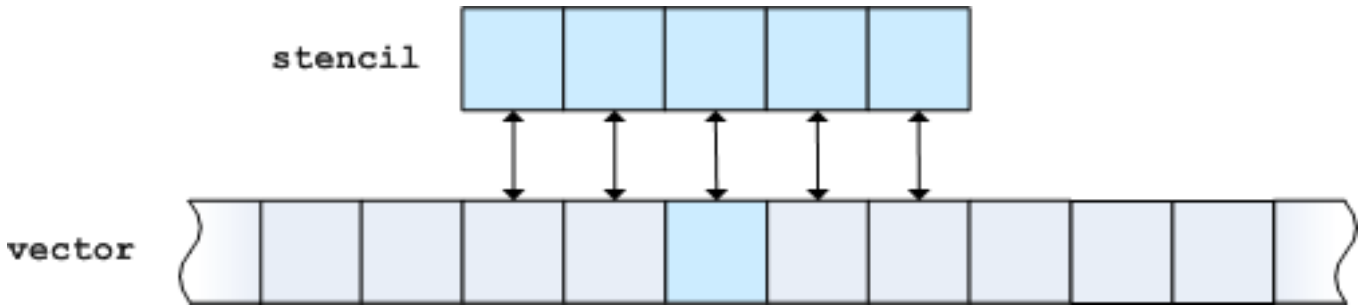
GPUlab DTU Informatics

A Finite Difference Example (II)

So in compact form we only need

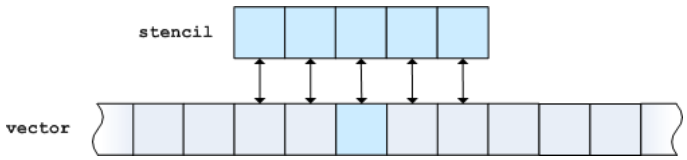
$$\mathbf{c} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}. \quad (1)$$

We call this the compact stencil.



It is embarrassingly parallel !

A Finite Difference Example (III)



Host version:

```
1  void finite_difference(float* out, float* in, float* stencil, int alpha, int N){
2      for(int n=alpha; n<N-alpha; ++n){
3          float sum = 0.f;
4          for(int i=-alpha; i<=alpha; ++i)
5              sum += stencil[alpha+i] * in[n+i];
6          out[n] = sum;
7      }
8  }
```

Device version:

```
1  __global__
2  void finite_difference(float* out, float* in, float* stencil, int alpha, int N){
3      int n = blockDim.x * blockIdx.x + threadIdx.x;
4      float sum = 0.f;
5      for(int i = -alpha; i<=alpha; ++i)
6          sum += stencil[alpha+i] * in[n+i];
7      out[n] = sum;
8  }
```

However, there is still some tweaking to do.

A Finite Difference Example (IV)

Performance results for CPU and GPU implementations, $\alpha = \beta$.

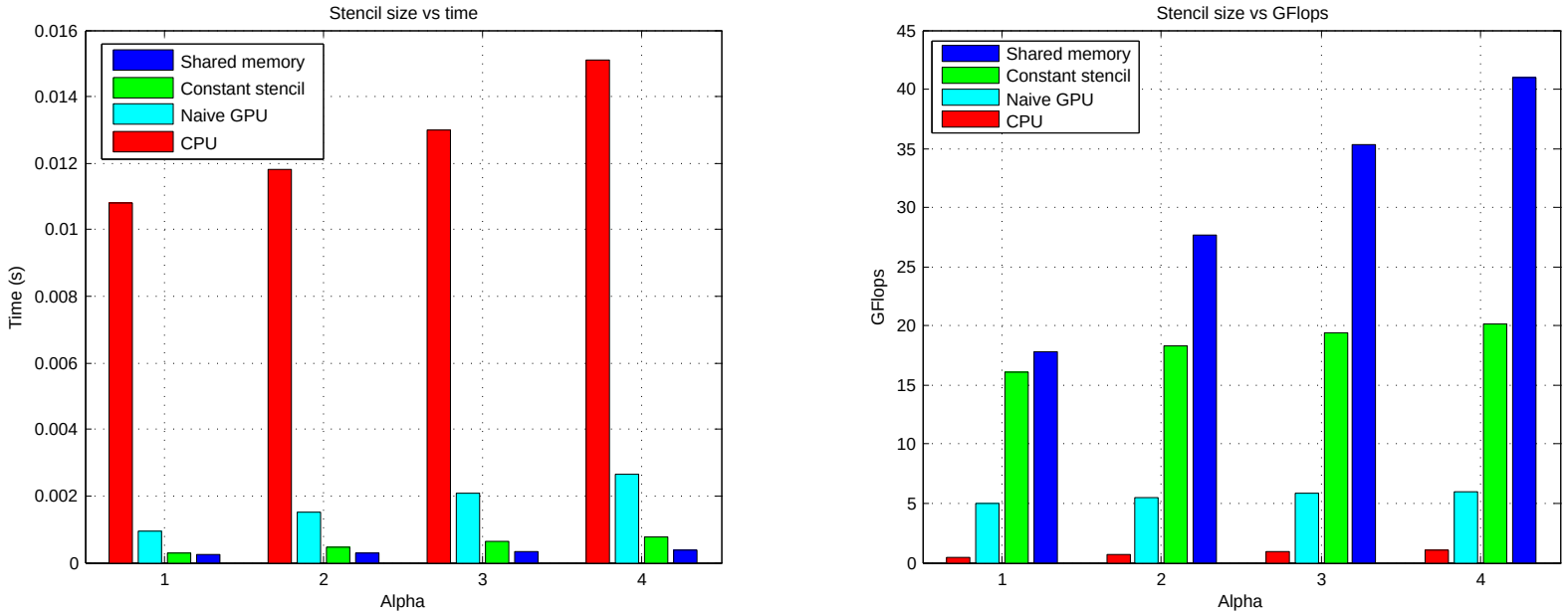


Figure: Timings for a vector with 1,000,000 elements. Using a Tesla C1070 GPU and an Intel Core i7 @ 1.73GHz CPU.

Framework Outline

Key components for PDE solvers

- Regular grid objects, 1D, 2D, 3D.

```
1  grid_dim<int> dim(100,100);                    // 100x100 grid
2  grid_dim<double> phys0(0.,0.);                 // Domain starts in x=0, y=0
3  grid_dim<double> phys1(1.,1.);                 // Domain end in x=1, y=1
4  grid_properties<int,double> grid_props(dim, phys0, phys1);
5  grid<double,device_memory> u(grid_props);      // Create u
6  grid<double,device_memory> f(grid_props);      // Create f
```

Framework Outline

Key components for PDE solvers

- Regular grid objects, 1D, 2D, 3D.
- Compact stencil-based *flexible order FD operators*

```
1  grid_dim<int> dim(100,100);                    // 100x100 grid
2  grid_dim<double> phys0(0.,0.);                 // Domain starts in x=0, y=0
3  grid_dim<double> phys1(1.,1.);                 // Domain end in x=1, y=1
4  grid_properties<int,double> grid_props(dim, phys0, phys1);
5  grid<double,device_memory> u(grid_props);      // Create u
6  grid<double,device_memory> f(grid_props);      // Create f
7
8  FD::stencil_2d<double> A(2,4);                 // Second order derivative, fourth order accuracy
9  A.matvec(u,f);                                 // Calculate f = du/dxx + du/dyy
```

Framework Outline

Key components for PDE solvers

- Regular grid objects, 1D, 2D, 3D.
- Compact stencil-based *flexible order FD operators*
- Iterative methods for solving large systems of eqs.

```
1  grid_dim<int> dim(100,100);                    // 100x100 grid
2  grid_dim<double> phys0(0.,0.);                 // Domain starts in x=0, y=0
3  grid_dim<double> phys1(1.,1.);                 // Domain end in x=1, y=1
4  grid_properties<int,double> grid_props(dim, phys0, phys1);
5  grid<double,device_memory> u(grid_props);      // Create u
6  grid<double,device_memory> f(grid_props);      // Create f
7
8  FD::stencil_2d<double> A(2,4);                 // Second order derivative, fourth order accuracy
9  A.matvec(u,f);                                 // Calculate f = du/dxx + du/dyy
10
11 monitor m(iter,rtol,atol);                     // Stopping criteria
12 solvers::cg cg_solver(A,m);                    // Create a CG solver from A
13 cg_solver.solve(u,f);                          // Solve Au = f
```

GPUlab DTU Informatics

Framework Outline

Key components for PDE solvers

- Regular grid objects, 1D, 2D, 3D.
- Compact stencil-based *flexible order FD operators*
- Iterative methods for solving large systems of eqs.
- Effective preconditioning strategies

A Fast Mixed-precision Strategy for Iterative GPU-based Solution of the Laplace Equation

Defect Correction Method

We found that the Defect Correction method works well for our Laplace problem

- High-order approximations (accuracy)
- Minimal storage overhead (problem size)
- Minimal global synchronization and reduction steps (parallelizable)
- Effective as GMRES in practice (effective)

Textbook Recipe

Algorithm: DC Method for approximate solution of $Ax = b$

1 Choose $x^{[0]}$ /* initial guess */
2 $k = 0$
3 **Repeat**
4 $r^{[k]} = b - Ax^{[k]}$ /* high order defect */
5 Solve $M\delta^{[k]} = r^{[k]}$ /* preconditioner */
6 $x^{[k+1]} = x^{[k]} + \delta^{[k]}$ /* defect correction */
7 $k = k + 1$
8 **Until** convergence or $k > k_{max}$

Defect Correction Method

We found that the Defect Correction method works well for our Laplace problem

- High-order approximations (accuracy)
- Minimal storage overhead (problem size)
- Minimal global synchronization and reduction steps (parallelizable)
- Effective as GMRES in practice (effective)

Textbook Recipe

Algorithm: DC Method for approximate solution of $Ax = b$

1 Choose $x^{[0]}$ /* initial guess */
2 $k = 0$
3 **Repeat**
4 $r^{[k]} = b - Ax^{[k]}$ /* high order defect */
5 Solve $M\delta^{[k]} = r^{[k]}$ /* preconditioner */
6 $x^{[k+1]} = x^{[k]} + \delta^{[k]}$ /* defect correction */
7 $k = k + 1$
8 **Until** convergence or $k > k_{max}$

Analysis of Defect Correction Convergence

Rewriting DC into the form of a stationary iterative method

$$x^{[k+1]} = x^{[k]} + \mathcal{M}^{-1}(b - \mathcal{A}x^{[k]}) \quad (2)$$

$$= (1 - \mathcal{M}^{-1}\mathcal{A})x^{[k]} + \mathcal{M}^{-1}b \quad (3)$$

$$= \mathcal{G}x^{[k]} + c, \quad k = 0, 1, \ldots \quad (4)$$

where $\mathcal{G}$ is called the iteration matrix. From stationary iterative theory we know that to ensure convergence towards the exact solution we must have

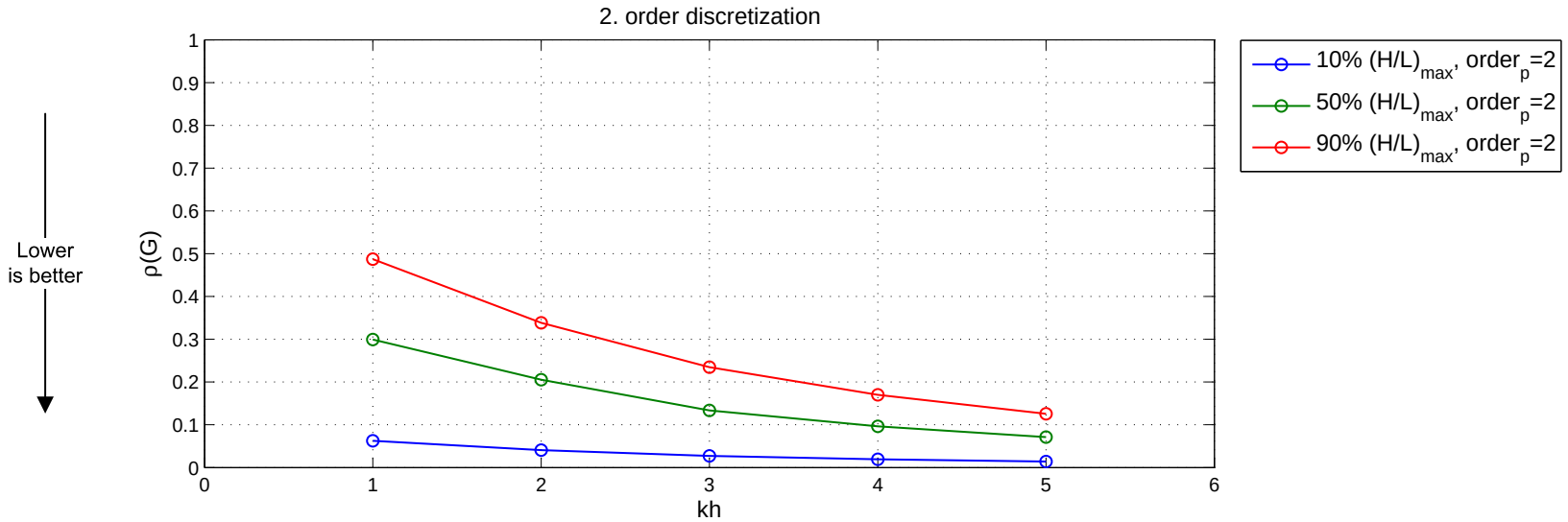
$$\rho(\mathcal{G}) < 1,$$

where $\rho(\mathcal{G})$ is the spectral radius of $\mathcal{G}$, i.e. the maximum absolute eigenvalue of $\mathcal{G}$. Closer to 0 means better convergence.

GPUlab DTU Informatics

Analysis of Defect Correction Convergence

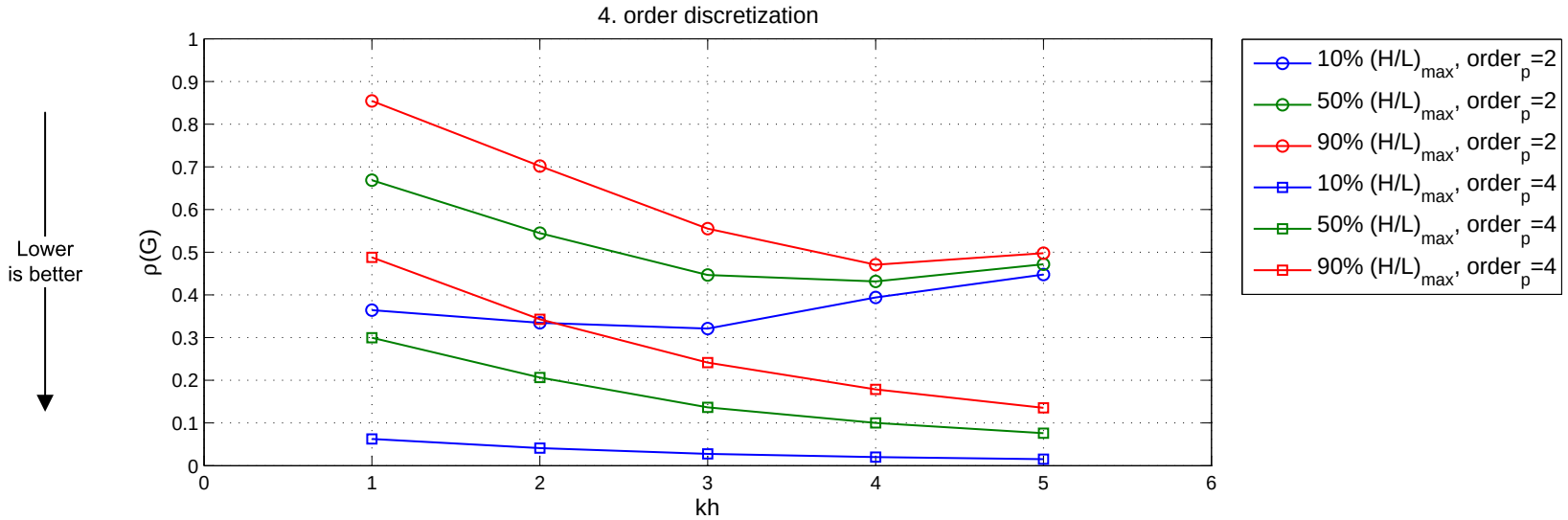
We can now predict attainable convergence rates for various free surface setups using linear flexible-order preconditioners.



Dispersion (kh) expresses ratio between water depth and wave length and influences to the condition number of the Laplacian matrix.

Analysis of Defect Correction Convergence

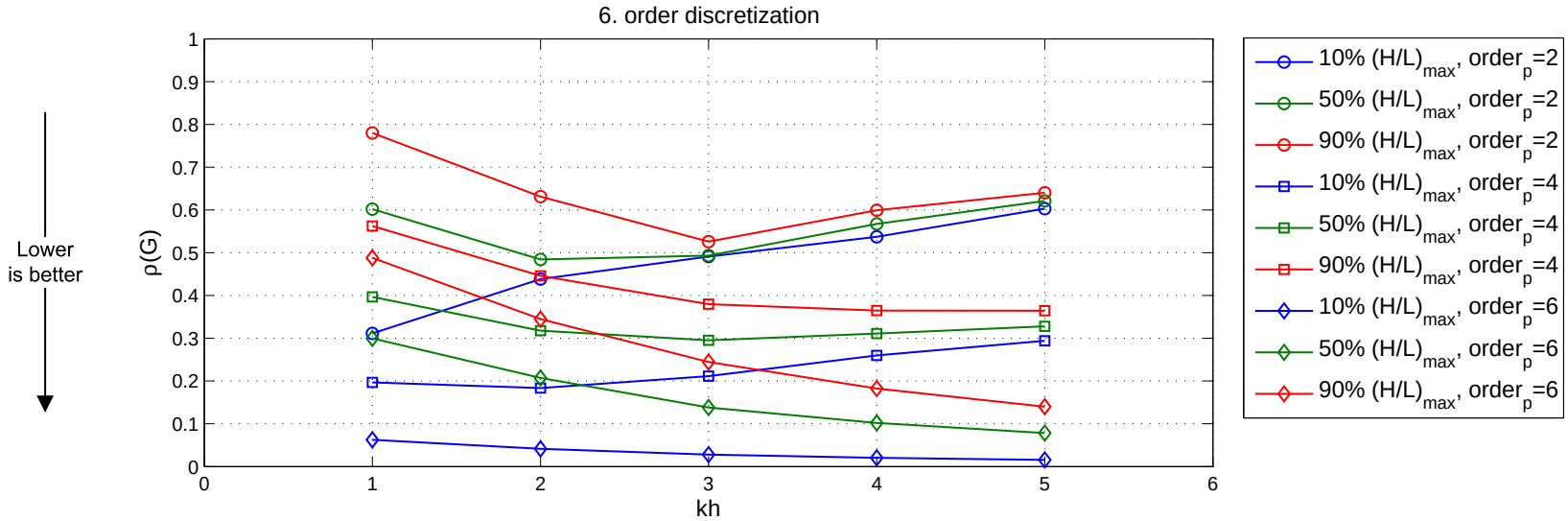
We can now predict attainable convergence rates for various free surface setups using linear flexible-order preconditioners.



Dispersion (kh) expresses ratio between water depth and wave length and influences to the condition number of the Laplacian matrix.

Analysis of Defect Correction Convergence

We can now predict attainable convergence rates for various free surface setups using linear flexible-order preconditioners.



Dispersion (kh) expresses ratio between water depth and wave length and influences to the condition number of the Laplacian matrix.

A Fast Mixed-precision Strategy for Iterative GPU-based Solution of the Laplace Equation

Mixed Precision

Definition

- An algorithm that mixes different machine precision numbers in its calculations – while maintaining a high precision solution.

Advantages

Bandwith bound

- 1 double = 2 floats = 64 bits
- Less storage - at all levels
- Less bandwith required

Compute bound

- 1 double multiplier $\approx$ 4 float multipliers
- 1 double adder $\approx$ 2 float adder
- On many GPUs 1:8

GPUlab DTU Informatics

Mixed Precision

Definition

- An algorithm that mixes different machine precision numbers in its calculations – while maintaining a high precision solution.

Question

- Can we obtain high accuracy solutions with low/fast precision calculations?

Note: Accuracy $\neq$ precision. 3.121872918723098 has good precision but is not an accurate representation of π.

Mixed Precision

Definition

- An algorithm that mixes different machine precision numbers in its calculations – while maintaining a high precision solution.

float s23e8

- s23e8 = 1 bit sign — 23 bit mantissa — 8 bit exponent
- $\pm d.dd\ldots d \times \beta^e$

The discrete set of floating point values are not uniform

Mixed Precision

Definition

- An algorithm that mixes different machine precision numbers in its calculations – while maintaining a high precision solution.

Roundoff error example

- Single precision roundoff error:

$$c = 0.5 + 0.5 + 0.000000004 - 0.000000003 = 1.000000001 = 1_{fl}$$

- Mixed precision fix:

$$a = 0.5 + 0.5 = 1_{fl}$$
$$b = 0.000000004 - 0.000000003 = 0.000000001_{fl}$$
$$c = a + b = 1.000000001_{dl}$$

GPUlab DTU Informatics

Mixed Precision Defect Correction

The same principle holds for the defect correction update – and all refinement processes in general.

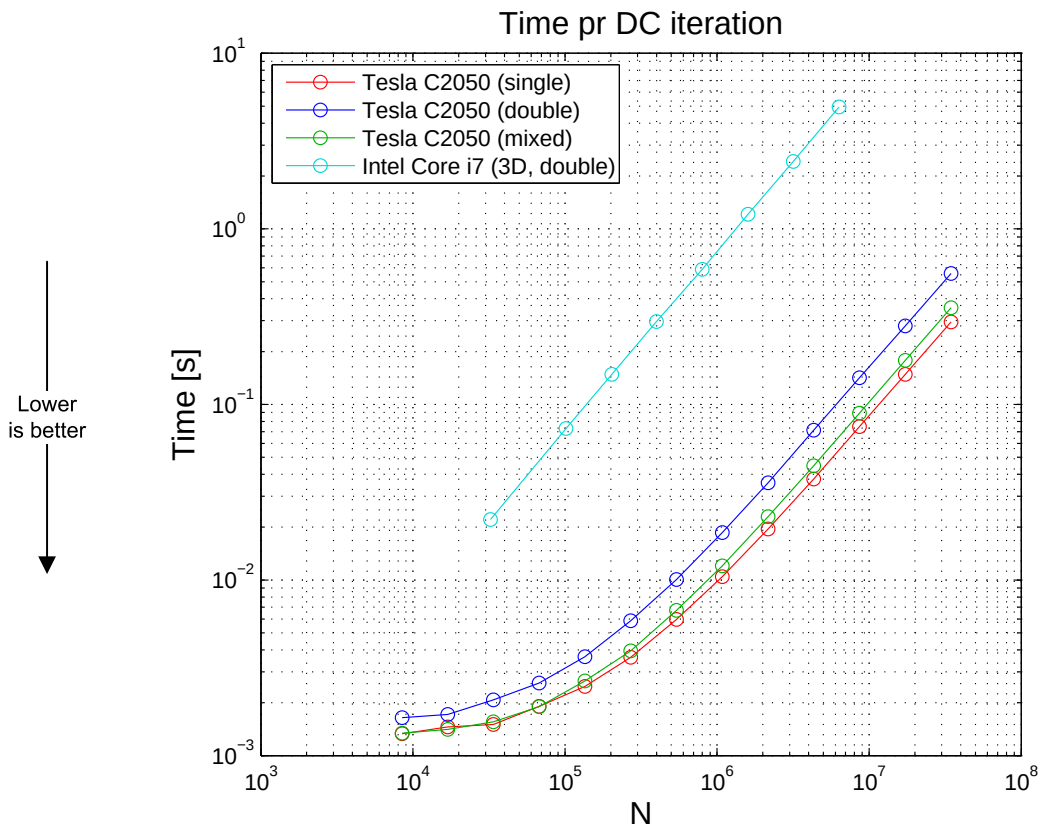
Mixed Precision DC

1 Choose $x^{[0]}$
2 $k = 0$
3 **Repeat**
4 $r^{[k]} = b - Ax^{[k]}$ /* Double Precision */
5 Solve $M\delta^{[k]} = r^{[k]}$ /* Single Precision */
6 $x^{[k+1]} = x^{[k]} + \delta^{[k]}$ /* Double Precision */
7 $k = k + 1$
8 **Until** convergence or $k > k_{max}$

Remember, much work lies within the preconditioner!

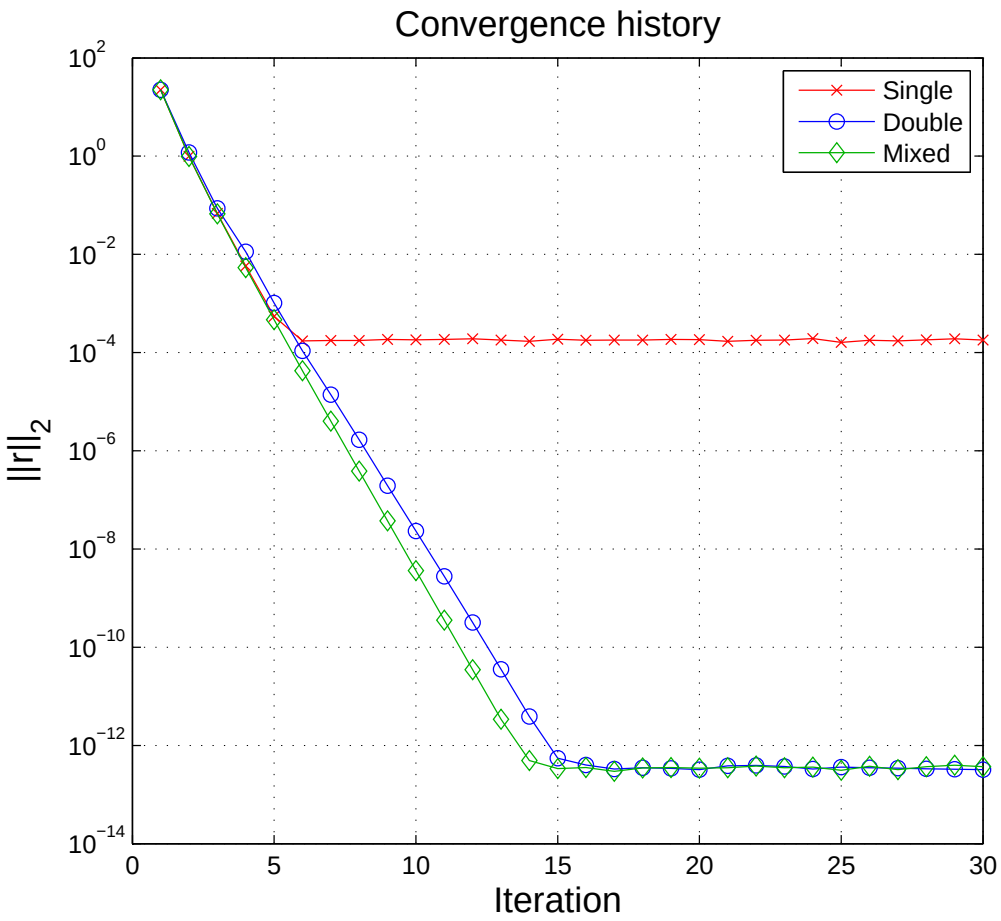
Mixed Precision GPU-based Performance Results

Timings per Defect Correction iteration. Using 6^{th} order accurate stencil, preconditioned with a linear 2^{nd} order accurate multigrid approach, DC+MG-RB-GS-1V(2,2).



Mixed Precision Convergence

The residual norm at every iteration confirms that the mixed precision algorithm in fact obtain high accuracy.



A Fast Mixed-precision Strategy for Iterative GPU-based Solution of the Laplace Equation

Allan P. Engsig-Karup.

Efficient low-storage solution of unsteady fully nonlinear water waves using a defect correction method.

Submitted to: Journal of Scientific Computing, 2011.

Allan Peter Engsig-Karup, Morten Gorm Madsen, and Stefan Lemvig Glimberg.

A massively parallel gpu-accelerated model for analysis of fully nonlinear free surface waves.

International Journal for Numerical Methods in Fluids, 2011.

GPUlab DTU Informatics